

EV316935375

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Mechanism for Obtaining and Applying Constraints to
Constructs within an Interactive Environment**

Inventor(s):

Jeffrey P. Snover

James W. Truher III

Kaushik Pushpavanam

Subramanian Viswanathan

ATTORNEY'S DOCKET NO. MS1-1740US

TECHNICAL FIELD

Subject matter disclosed herein relates to interactive environments, and in particular to obtaining and applying constraints within an interactive environment.

BACKGROUND OF THE INVENTION

In general, there are two types of code: compiled code and interpreted code. In the past, compiled code was compiled into an object code and then linked with other object codes to create an executable that was executed at run-time. Today, in some environments, compiled code includes source code that has been compiled into an intermediate form. At run-time, the intermediate form is compiled into native code for execution. In either of these scenarios, a developer may specify a type for each construct programmed in the source code. Types include integer, string, float, and the like. In contrast, for interpreted code within an interactive environment, the interactive environment processes each variable as a string. Therefore, interactive users may not specify the type for a variable.

Therefore, there is a need for a mechanism for assigning types and other constraints to variables in an interactive environment.

SUMMARY OF THE INVENTION

The present mechanism obtains constraints within an interactive environment, associates these constraints with constructs, and then applies these constraints to the constructs when encountering the constructs. The constraints may be saved in metadata associated with the respective construct. The constraints may specify a data type for the construct, a predicate directive, a documentation directive, a parsing directive, a data generation directive, a data

1 validation directive, or an object processing and encoding directive. The
2 constraints are extendable to support other directives. The mechanism allows
3 interactive users to easily specify constraints interactively.

4 BRIEF DESCRIPTION OF THE DRAWINGS

5 FIGURE 1 illustrates an exemplary computing device that may use an
6 exemplary administrative tool environment.

7 FIGURE 2 is a block diagram generally illustrating an overview of an
8 exemplary administrative tool framework for the present administrative tool
9 environment.

10 FIGURE 3 is a block diagram illustrating components within the host-
11 specific components of the administrative tool framework shown in FIGURE 2.

12 FIGURE 4 is a block diagram illustrating components within the core
13 engine component of the administrative tool framework shown in FIGURE 2.

14 FIGURE 5 is one exemplary data structure for specifying a cmdlet suitable
15 for use within the administrative tool framework shown in FIGURE 2.

16 FIGURE 6 is an exemplary data structure for specifying a command base
17 type from which a cmdlet shown in FIGURE 5 is derived.

18 FIGURE 7 is another exemplary data structure for specifying a cmdlet
19 suitable for use within the administrative tool framework shown in FIGURE 2.

20 FIGURE 8 is a logical flow diagram illustrating an exemplary process for
21 host processing that is performed within the administrative tool framework shown
22 in FIGURE 2.

23 FIGURE 9 is a logical flow diagram illustrating an exemplary process for
24 handling input that is performed within the administrative tool framework shown
25 in FIGURE 2.

1 FIGURE 10 is a logical flow diagram illustrating a process for processing
2 scripts suitable for use within the process for handling input shown in FIGURE 9.

3 FIGURE 11 is a logical flow diagram illustrating a script pre-processing
4 process suitable for use within the script processing process shown in FIGURE 10.

5 FIGURE 12 is a logical flow diagram illustrating a process for applying
6 constraints suitable for use within the script processing process shown in FIGURE
7 10.

8 FIGURE 13 is a functional flow diagram illustrating the processing of a
9 command string in the administrative tool framework shown in FIGURE 2.

10 FIGURE 14 is a logical flow diagram illustrating a process for processing
11 commands strings suitable for use within the process for handling input shown in
12 FIGURE 9.

13 FIGURE 15 is a logical flow diagram illustrating an exemplary process for
14 creating an instance of a cmdlet suitable for use within the processing of command
15 strings shown in FIGURE 14.

16 FIGURE 16 is a logical flow diagram illustrating an exemplary process for
17 populating properties of a cmdlet suitable for use within the processing of
18 commands shown in FIGURE 14.

19 FIGURE 17 is a logical flow diagram illustrating an exemplary process for
20 executing the cmdlet suitable for use within the processing of commands shown in
21 FIGURE 14.

22 FIGURE 18 is a functional block diagram of an exemplary extended type
23 manager suitable for use within the administrative tool framework shown in
24 FIGURE 2.

1 FIGURE 19 graphically depicts exemplary sequences for output processing
2 cmdlets within a pipeline.

3 FIGURE 20 illustrates exemplary processing performed by one of the
4 output processing cmdlets shown in FIGURE 19.

5 FIGURE 21 graphically depicts an exemplary structure for display
6 information accessed during the processing of FIGURE 20.

7 FIGURE 22 is a table listing an exemplary syntax for exemplary output
8 processing cmdlets.

9 FIGURE 23 illustrates results rendered by the out/console cmdlet using
10 various pipeline sequences of the output processing cmdlets.

11 DETAILED DESCRIPTION

12 Briefly stated, the present mechanism obtains constraints within an
13 interactive environment and then applies these constraints to constructs entered
14 within the interactive environment. The constraints may be saved in metadata
15 associated with the respective construct. The constraints may specify a data type
16 for the construct, a valid range for the construct, and the like. The mechanism
17 allows interactive users to easily specify constraints interactively.

18 The following description sets forth a specific exemplary administrative
19 tool environment in which the mechanism operates. Other exemplary
20 environments may include features of this specific embodiment and/or other
21 features, which aim to facilitate constraint processing within an interactive
22 environment.

23 The following detailed description is divided into several sections. A first
24 section describes an illustrative computing environment in which the
25 administrative tool environment may operate. A second section describes an

1 exemplary framework for the administrative tool environment. Subsequent
2 sections describe individual components of the exemplary framework and the
3 operation of these components. For example, the section on "Exemplary
4 Processing of Scripts", in conjunction with FIGURE 12, describes an exemplary
5 mechanism for obtaining and applying constraints in an interactive environment.

6 Exemplary Computing Environment

7 FIGURE 1 illustrates an exemplary computing device that may be used in
8 an exemplary administrative tool environment. In a very basic configuration,
9 computing device 100 typically includes at least one processing unit 102 and
10 system memory 104. Depending on the exact configuration and type of
11 computing device, system memory 104 may be volatile (such as RAM), non-
12 volatile (such as ROM, flash memory, etc.) or some combination of the two.
13 System memory 104 typically includes an operating system 105, one or more
14 program modules 106, and may include program data 107. The operating system
15 106 include a component-based framework 120 that supports components
16 (including properties and events), objects, inheritance, polymorphism, reflection,
17 and provides an object-oriented component-based application programming
18 interface (API), such as that of the .NETTM Framework manufactured by
19 Microsoft Corporation, Redmond, WA. The operating system 105 also includes
20 an administrative tool framework 200 that interacts with the component-based
21 framework 120 to support development of administrative tools (not shown). This
22 basic configuration is illustrated in FIGURE 1 by those components within dashed
23 line 108.

1 Computing device 100 may have additional features or functionality. For
2 example, computing device 100 may also include additional data storage devices
3 (removable and/or non-removable) such as, for example, magnetic disks, optical
4 disks, or tape. Such additional storage is illustrated in FIGURE 1 by removable
5 storage 109 and non-removable storage 110. Computer storage media may
6 include volatile and nonvolatile, removable and non-removable media
7 implemented in any method or technology for storage of information, such as
8 computer readable instructions, data structures, program modules, or other data.
9 System memory 104, removable storage 109 and non-removable storage 110 are
10 all examples of computer storage media. Computer storage media includes, but is
11 not limited to, RAM, ROM, EEPROM, flash memory or other memory
12 technology, CD-ROM, digital versatile disks (DVD) or other optical storage,
13 magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage
14 devices, or any other medium which can be used to store the desired information
15 and which can be accessed by computing device 100. Any such computer storage
16 media may be part of device 100. Computing device 100 may also have input
17 device(s) 112 such as keyboard, mouse, pen, voice input device, touch input
18 device, etc. Output device(s) 114 such as a display, speakers, printer, etc. may
19 also be included. These devices are well known in the art and need not be
20 discussed at length here.

21 Computing device 100 may also contain communication connections 116
22 that allow the device to communicate with other computing devices 118, such as
23 over a network. Communication connections 116 are one example of
24 communication media. Communication media may typically be embodied by
25

1 computer readable instructions, data structures, program modules, or other data in
2 a modulated data signal, such as a carrier wave or other transport mechanism, and
3 includes any information delivery media. The term “modulated data signal”
4 means a signal that has one or more of its characteristics set or changed in such a
5 manner as to encode information in the signal. By way of example, and not
6 limitation, communication media includes wired media such as a wired network or
7 direct-wired connection, and wireless media such as acoustic, RF, infrared and
8 other wireless media. The term computer readable media as used herein includes
9 both storage media and communication media.

10 Exemplary Administrative Tool Framework

11 FIGURE 2 is a block diagram generally illustrating an overview of an
12 exemplary administrative tool framework 200. Administrative tool framework
13 200 includes one or more host components 202, host-specific components 204,
14 host-independent components 206, and handler components 208. The host-
15 independent components 206 may communicate with each of the other
16 components (i.e., the host components 202, the host-specific components 204, and
17 the handler components 208). Each of these components are briefly described
18 below and described in further detail, as needed, in subsequent sections.

19 Host components

20 The host components 202 include one or more host programs (e.g., host
21 programs 210-214) that expose automation features for an associated application
22 to users or to other programs. Each host program 210-214 may expose these
23 automation features in its own particular style, such as via a command line, a
24 graphical user interface (GUI), a voice recognition interface, application
25

1 programming interface (API), a scripting language, a web service, and the like.
2 However, each of the host programs 210-214 expose the one or more automation
3 features through a mechanism provided by the administrative tool framework.

4 In this example, the mechanism uses cmdlets to surface the administrative
5 tool capabilities to a user of the associated host program 210-214. In addition, the
6 mechanism uses a set of interfaces made available by the host to embed the
7 administrative tool environment within the application associated with the
8 corresponding host program 210-214. Throughout the following discussion, the
9 term "cmdlet" is used to refer to commands that are used within the exemplary
10 administrative tool environment described with reference to FIGURES 2-23.

11 Cmdlets correspond to commands in traditional administrative
12 environments. However, cmdlets are quite different than these traditional
13 commands. For example, cmdlets are typically smaller in size than their
14 counterpart commands because the cmdlets can utilize common functions
15 provided by the administrative tool framework, such as parsing, data validation,
16 error reporting, and the like. Because such common functions can be implemented
17 once and tested once, the use of cmdlets throughout the administrative tool
18 framework allows the incremental development and test costs associated with
19 application-specific functions to be quite low compared to traditional
20 environments.

21 In addition, in contrast to traditional environments, cmdlets do not need to
22 be stand-alone executable programs. Rather, cmdlets may run in the same
23 processes within the administrative tool framework. This allows cmdlets to
24 exchange "live" objects between each other. This ability to exchange "live"
25

1 objects allows the cmdlets to directly invoke methods on these objects. The
2 details for creating and using cmdlets are described in further detail below.

3 In overview, each host program 210-214 manages the interactions between
4 the user and the other components within the administrative tool framework.
5 These interactions may include prompts for parameters, reports of errors, and the
6 like. Typically, each host program 210-213 may provide its own set of specific
7 host cmdlets (e.g., host cmdlets 218). For example, if the host program is an email
8 program, the host program may provide host cmdlets that interact with mailboxes
9 and messages. Even though FIGURE 2 illustrates host programs 210-214, one
10 skilled in the art will appreciate that host components 202 may include other host
11 programs associated with existing or newly created applications. These other host
12 programs will also embed the functionality provided by the administrative tool
13 environment within their associated application. The processing provided by a
14 host program is described in detail below in conjunction with FIGURE 8.

15 In the examples illustrated in FIGURE 2, a host program may be a
16 management console (i.e., host program 210) that provides a simple, consistent,
17 administration user interface for users to create, save, and open administrative
18 tools that manage the hardware, software, and network components of the
19 computing device. To accomplish these functions, host program 210 provides a
20 set of services for building management GUIs on top of the administrative tool
21 framework. The GUI interactions may also be exposed as user-visible scripts that
22 help teach the users the scripting capabilities provided by the administrative tool
23 environment.
24
25

1 In another example, the host program may be a command line interactive
2 shell (i.e., host program 212). The command line interactive shell may allow shell
3 metadata 216 to be input on the command line to affect processing of the
4 command line.

5 In still another example, the host program may be a web service (i.e., host
6 program 214) that uses industry standard specifications for distributed computing
7 and interoperability across platforms, programming languages, and applications.

8 In addition to these examples, third parties may add their own host
9 components by creating "third party" or "provider" interfaces and provider
10 cmdlets that are used with their host program or other host programs. The
11 provider interface exposes an application or infrastructure so that the application
12 or infrastructure can be manipulated by the administrative tool framework. The
13 provider cmdlets provide automation for navigation, diagnostics, configuration,
14 lifecycle, operations, and the like. The provider cmdlets exhibit polymorphic
15 cmdlet behavior on a completely heterogeneous set of data stores. The
16 administrative tool environment operates on the provider cmdlets with the same
17 priority as other cmdlet classes. The provider cmdlet is created using the same
18 mechanisms as the other cmdlets. The provider cmdlets expose specific
19 functionality of an application or an infrastructure to the administrative tool
20 framework. Thus, through the use of cmdlets, product developers need only create
21 one host component that will then allow their product to operate with many
22 administrative tools. For example, with the exemplary administrative tool
23 environment, system level graphical user interface help menus may be integrated
24 and ported to existing applications.
25

Host-specific components

The host-specific components 204 include a collection of services that computing systems (e.g., computing device 100 in FIGURE 1) use to isolate the administrative tool framework from the specifics of the platform on which the framework is running. Thus, there is a set of host-specific components for each type of platform. The host-specific components allow the users to use the same administrative tools on different operating systems.

Turning briefly to FIGURE 3, the host-specific components 204 may include an intellisense/metadata access component 302, a help cmdlet component 304, a configuration/registration component 306, a cmdlet setup component 308, and an output interface component 309. Components 302-308 communicate with a database store manager 312 associated with a database store 314. The parser 220 and script engine 222 communicate with the intellisense/metadata access component 302. The core engine 224 communicates with the help cmdlet component 304, the configuration/registration component 306, the cmdlet setup component 308, and the output interface component 309. The output interface component 309 includes interfaces provided by the host to out cmdlets. These out cmdlets can then call the host's output object to perform the rendering. Host-specific components 204 may also include a logging/auditing component 310, which the core engine 224 uses to communicate with host specific (i.e., platform specific) services that provide logging and auditing capabilities.

In one exemplary administrative tool framework, the intellisense/metadata access component 302 provides auto-completion of commands, parameters, and

1 parameter values. The help cmdlet component 304 provides a customized help
2 system based on a host user interface.

3 Handler components

4 Referring back to FIGURE 2, the handler components 208 includes legacy
5 utilities 230, management cmdlets 232, non-management cmdlets 234, remoting
6 cmdlets 236, and a web service interface 238. The management cmdlets 232 (also
7 referred to as platform cmdlets) include cmdlets that query or manipulate the
8 configuration information associated with the computing device. Because
9 management cmdlets 232 manipulate system type information, they are dependant
10 upon a particular platform. However, each platform typically has management
11 cmdlets 232 that provide similar actions as management cmdlets 232 on other
12 platforms. For example, each platform supports management cmdlets 232 that get
13 and set system administrative attributes (e.g., get/process, set/IPAddress). The
14 host-independent components 206 communicate with the management cmdlets via
15 cmdlet objects generated within the host-independent components 206.
16 Exemplary data structures for cmdlets objects will be described in detail below in
17 conjunction with FIGURES 5-7.

18 The non-management cmdlets 234 (sometimes referred to as base cmdlets)
19 include cmdlets that group, sort, filter, and perform other processing on objects
20 provided by the management cmdlets 232. The non-management cmdlets 234
21 may also include cmdlets for formatting and outputting data associated with the
22 pipelined objects. An exemplary mechanism for providing a data driven command
23 line output is described below in conjunction with FIGURES 19-23. The non-
24 management cmdlets 234 may be the same on each platform and provide a set of
25

1 utilities that interact with host-independent components 206 via cmdlet objects.
2 The interactions between the non-management cmdlets 234 and the host-
3 independent components 206 allow reflection on objects and allow processing on
4 the reflected objects independent of their (object) type. Thus, these utilities allow
5 developers to write non-management cmdlets once and then apply these non-
6 management cmdlets across all classes of objects supported on a computing
7 system. In the past, developers had to first comprehend the format of the data that
8 was to be processed and then write the application to process only that data. As a
9 consequence, traditional applications could only process data of a very limited
10 scope. One exemplary mechanism for processing objects independent of their
11 object type is described below in conjunction with FIGURE 18.

12 The legacy utilities 230 include existing executables, such as win32
13 executables that run under cmd.exe. Each legacy utility 230 communicates with
14 the administrative tool framework using text streams (i.e., stdin and stdout), which
15 are a type of object within the object framework. Because the legacy utilities 230
16 utilize text streams, reflection-based operations provided by the administrative tool
17 framework are not available. The legacy utilities 230 execute in a different
18 process than the administrative tool framework. Although not shown, other
19 cmdlets may also operate out of process.

20 The remoting cmdlets 236, in combination with the web service interface
21 238, provide remoting mechanisms to access interactive and programmatic
22 administrative tool environments on other computing devices over a
23 communication media, such as internet or intranet (e.g., internet/intranet 240
24 shown in FIGURE 2). In one exemplary administrative tool framework, the
25

1 remoting mechanisms support federated services that depend on infrastructure that
2 spans multiple independent control domains. The remoting mechanism allows
3 scripts to execute on remote computing devices. The scripts may be run on a
4 single or on multiple remote systems. The results of the scripts may be processed
5 as each individual script completes or the results may be aggregated and processed
6 en-masse after all the scripts on the various computing devices have completed.

7 For example, web service 214 shown as one of the host components 202
8 may be a remote agent. The remote agent handles the submission of remote
9 command requests to the parser and administrative tool framework on the target
10 system. The remoting cmdlets serve as the remote client to provide access to the
11 remote agent. The remote agent and the remoting cmdlets communicate via a
12 parsed stream. This parsed stream may be protected at the protocol layer, or
13 additional cmdlets may be used to encrypt and then decrypt the parsed stream.

14 Host-independent components

15 The host-independent components 206 include a parser 220, a script engine
16 222 and a core engine 224. The host-independent components 206 provide
17 mechanisms and services to group multiple cmdlets, coordinate the operation of
18 the cmdlets, and coordinate the interaction of other resources, sessions, and jobs
19 with the cmdlets.

20 Exemplary Parser

21 The parser 220 provides mechanisms for receiving input requests from
22 various host programs and mapping the input requests to uniform cmdlet objects
23 that are used throughout the administrative tool framework, such as within the
24 core engine 224. In addition, the parser 220 may perform data processing based
25

1 on the input received. One exemplary method for performing data processing
2 based on the input is described below in conjunction with FIGURE 12. The
3 parser 220 of the present administrative tool framework provides the capability to
4 easily expose different languages or syntax to users for the same capabilities. For
5 example, because the parser 220 is responsible for interpreting the input requests,
6 a change to the code within the parser 220 that affects the expected input syntax
7 will essentially affect each user of the administrative tool framework. Therefore,
8 system administrators may provide different parsers on different computing
9 devices that support different syntax. However, each user operating with the same
10 parser will experience a consistent syntax for each cmdlet. In contrast, in
11 traditional environments, each command implemented its own syntax. Thus, with
12 thousands of commands, each environment supported several different syntax,
13 usually many of which were inconsistent with each other.

14 Exemplary Script Engine

15 The script engine 222 provides mechanisms and services to tie multiple
16 cmdlets together using a script. A script is an aggregation of command lines that
17 share session state under strict rules of inheritance. The multiple command lines
18 within the script may be executed either synchronously or asynchronously, based
19 on the syntax provided in the input request. The script engine 222 has the ability
20 to process control structures, such as loops and conditional clauses and to process
21 variables within the script. The script engine also manages session state and gives
22 cmdlets access to session data based on a policy (not shown).

23 Exemplary Core Engine

24
25

1 The core engine 224 is responsible for processing cmdlets identified by the
2 parser 220. Turning briefly to FIGURE 4, an exemplary core engine 224 within
3 the administrative tool framework 200 is illustrated. The exemplary core engine
4 224 includes a pipeline processor 402, a loader 404, a metadata processor 406, an
5 error & event handler 408, a session manager 410, and an extended type manager
6 412.

7 Exemplary Metadata Processor

8 The metadata processor 406 is configured to access and store metadata
9 within a metadata store, such as database store 314 shown in FIGURE 3. The
10 metadata may be supplied via the command line, within a cmdlet class definition,
11 and the like. Different components within the administrative tool framework 200
12 may request the metadata when performing their processing. For example, parser
13 202 may request metadata to validate parameters supplied on the command line.

14 Exemplary Error & Event Processor

15 The error & event processor 408 provides an error object to store
16 information about each occurrence of an error during processing of a command
17 line. For additional information about one particular error and event processor
18 which is particularly suited for the present administrative tool framework, refer to
19 U.S. Patent Application No. ____/ U.S. Patent No. ____, entitled "System and
20 Method for Persisting Error Information in a Command Line Environment", which
21 is owned by the same assignee as the present invention, and is incorporated here
22 by reference.

23 Exemplary Session Manager

24 The session manager 410 supplies session and state information to other
25 components within the administrative tool framework 200. The state information

1 managed by the session manager may be accessed by any cmdlet, host, or core
2 engine via programming interfaces. These programming interfaces allow for the
3 creation, modification, and deletion of state information.

4 Exemplary Pipeline Processor and Loader

5 The loader 404 is configured to load each cmdlet in memory in order for
6 the pipeline processor 402 to execute the cmdlet. The pipeline processor 402
7 includes a cmdlet processor 420 and a cmdlet manager 422. The cmdlet
8 processor 420 dispatches individual cmdlets. If the cmdlet requires execution on a
9 remote, or a set of remote machines, the cmdlet processor 420 coordinates the
10 execution with the remoting cmdlet 236 shown in FIGURE 2. The cmdlet
11 manager 422 handles the execution of aggregations of cmdlets. The cmdlet
12 manager 422, the cmdlet processor 420, and the script engine 222 (FIGURE 2)
13 communicate with each other in order to perform the processing on the input
14 received from the host program 210-214. The communication may be recursive in
15 nature. For example, if the host program provides a script, the script may invoke
16 the cmdlet manager 422 to execute a cmdlet, which itself may be a script. The
17 script may then be executed by the script engine 222. One exemplary process
18 flow for the core engine is described in detail below in conjunction with FIGURE
19 14.

20 Exemplary Extended Type Manager

21 As mentioned above, the administrative tool framework provides a set of
22 utilities that allows reflection on objects and allows processing on the reflected
23 objects independent of their (object) type. The administrative tool framework 200
24 interacts with the component framework on the computing system (component
25 framework 120 in FIGURE 1) to perform this reflection. As one skilled in the art

1 will appreciate, reflection provides the ability to query an object and to obtain a
2 type for the object, and then reflect on various objects and properties associated
3 with that type of object to obtain other objects and/or a desired value.

4 Even though reflection provides the administrative tool framework 200 a
5 considerable amount of information on objects, the inventors appreciated that
6 reflection focuses on the type of object. For example, when a database datatable is
7 reflected upon, the information that is returned is that the datatable has two
8 properties: a column property and a row property. These two properties do not
9 provide sufficient detail regarding the "objects" within the datatable. Similar
10 problems arise when reflection is used on extensible markup language (XML) and
11 other objects.

12 Thus, the inventors conceived of an extended type manager 412 that
13 focuses on the usage of the type. For this extended type manager, the type of
14 object is not important. Instead, the extended type manager is interested in
15 whether the object can be used to obtain required information. Continuing with
16 the above datatable example, the inventors appreciated that knowing that the
17 datatable has a column property and a row property is not particularly interesting,
18 but appreciated that one column contained information of interest. Focusing on
19 the usage, one could associate each row with an "object" and associate each
20 column with a "property" of that "object". Thus, the extended type manager 412
21 provides a mechanism to create "objects" from any type of precisely parse-able
22 input. In so doing, the extended type manager 412 supplements the reflection
23 capabilities provided by the component-based framework 120 and extends
24 "reflection" to any type of precisely parse-able input.
25

1 In overview, the extended type manager is configured to access precisely
2 parse-able input (not shown) and to correlate the precisely parse-able input with a
3 requested data type. The extended type manager 412 then provides the requested
4 information to the requesting component, such as the pipeline processor 402 or
5 parser 220. In the following discussion, precisely parse-able input is defined as
6 input in which properties and values may be discerned. Some exemplary precisely
7 parse-able input include Windows Management Instrumentation (WMI) input,
8 ActiveX Data Objects (ADO) input, eXtensible Markup Language (XML) input,
9 and object input, such as .NET objects. Other precisely parse-able input may
10 include third party data formats.

11 Turning briefly to FIGURE 18, a functional block diagram of an exemplary
12 extended type manager for use within the administrative tool framework is shown.
13 For explanation purposes, the functionality (denoted by the number "3" within a
14 circle) provided by the extended type manager is contrasted with the functionality
15 provided by a traditional tightly bound system (denoted by the number "1" within
16 a circle) and the functionality provided by a reflection system (denoted by the
17 number "2" within a circle). In the traditional tightly bound system, a caller 1802
18 within an application directly accesses the information (e.g., properties P1 and P2,
19 methods M1 and M2) within object A. As mentioned above, the caller 1802 must
20 know, a priori, the properties (e.g., properties P1 and P2) and methods (e.g.,
21 methods M1 and M2) provided by object A at compile time. In the reflection
22 system, generic code 1820 (not dependent on any data type) queries a system 1808
23 that performs reflection 1810 on the requested object and returns the information
24 (e.g., properties P1 and P2, methods M1 and M2) about the object (e.g., object A)
25

1 to the generic code 1820. Although not shown in object A, the returned
2 information may include additional information, such as vendor, file, date, and the
3 like. Thus, through reflection, the generic code 1820 obtains at least the same
4 information that the tightly bound system provides. The reflection system also
5 allows the caller 1802 to query the system and get additional information without
6 any a priori knowledge of the parameters.

7 In both the tightly bound systems and the reflection systems, new data
8 types can not be easily incorporated within the operating environment. For
9 example, in a tightly bound system, once the operating environment is delivered,
10 the operating environment can not incorporate new data types because it would
11 have to be rebuilt in order to support them. Likewise, in reflection systems, the
12 metadata for each object class is fixed. Thus, incorporating new data types is not
13 usually done.

14 However, with the present extended type manager new data types can be
15 incorporated into the operating system. With the extended type manager 1822,
16 generic code 1820 may reflect on a requested object to obtain extended data types
17 (e.g., object A') provided by various external sources, such as a third party objects
18 (e.g., object A' and B), a semantic web 1832, an ontology service 1834, and the
19 like. As shown, the third party object may extend an existing object (e.g., object
20 A') or may create an entirely new object (e.g., object B).

21 Each of these external sources may register their unique structure within a
22 type metadata 1840 and may provide code 1842. When an object is queried, the
23 extended type manager reviews the type metadata 1840 to determine whether the
24 object has been registered. If the object is not registered within the type metadata
25 1840, reflection is performed. Otherwise, extended reflection is performed. The

code 1842 returns the additional properties and methods associated with the type being reflected upon. For example, if the input type is XML, the code 1842 may include a description file that describes the manner in which the XML is used to create the objects from the XML document. Thus, the type metadata 1840 describes how the extended type manager 412 should query various types of precisely parse-able input (e.g., third party objects A' and B, semantic web 1832) to obtain the desired properties for creating an object for that specific input type and the code 1842 provides the instructions to obtain these desired properties. As a result, the extended type manager 412 provides a layer of indirection that allows "reflection" on all types of objects.

In addition to providing extended types, the extend type manager 412 provides additional query mechanisms, such as a property path mechanism, a key mechanism, a compare mechanism, a conversion mechanism, a globber mechanism, a property set mechanism, a relationship mechanism, and the like. Each of these query mechanisms, described below in the section "Exemplary Extended Type Manager Processing", provides flexibility to system administrators when entering command strings. Various techniques may be used to implement the semantics for the extended type manager. Three techniques are described below. However, those skilled in the art will appreciate that variations of these techniques may be used without departing from the scope of the claimed invention.

In one technique, a series of classes having static methods (e.g., `getproperty()`) may be provided. An object is input into the static method (e.g., `getproperty(object)`), and the static method returns a set of results. In another technique, the operating environment envelopes the object with an adapter. Thus,

no input is supplied. Each instance of the adapter has a `getproperty` method that acts upon the enveloped object and returns the properties for the enveloped object.

The following is pseudo code illustrating this technique:

```
Class Adaptor
```

```
{
```

```
    Object X;
```

```
    getProperties();
```

```
};
```

In still another technique, an adaptor class subclasses the object. Traditionally, subclassing occurred before compilation. However, with certain operating environments, subclassing may occur dynamically. For these types of environments, the following is pseudo code illustrating this technique:

```
Class Adaptor : A
```

```
{
```

```
    getProperties()
```

```
    {
```

```
        return data;
```

```
    }
```

```
};
```

Thus, as illustrated in FIGURE 18, the extended type manager allows developers to create a new data type, register the data type, and allow other applications and cmdlets to use the new data type. In contrast, in prior

1 administrative environments, each data type had to be known at compile time so
2 that a property or method associated with an object instantiated from that data type
3 could be directly accessed. Therefore, adding new data types that were supported
4 by the administrative environment was seldom done in the past.

5 Referring back to FIGURE 2, in overview, the administrative tool
6 framework 200 does not rely on the shell for coordinating the execution of
7 commands input by users, but rather, splits the functionality into processing
8 portions (e.g., host-independent components 206) and user interaction portions
9 (e.g., via host cmdlets). In addition, the present administrative tool environment
10 greatly simplifies the programming of administrative tools because the code
11 required for parsing and data validation is no longer included within each
12 command, but is rather provided by components (e.g., parser 220) within the
13 administrative tool framework. The exemplary processing performed within the
14 administrative tool framework is described below.

15 Exemplary Operation

16 FIGURES 5-7 graphically illustrate exemplary data structures used within
17 the administrative tool environment. FIGURES 8-17 graphically illustrate
18 exemplary processing flows within the administrative tool environment. One
19 skilled in the art will appreciate that certain processing may be performed by a
20 different component than the component described below without departing from
21 the scope of the present invention. Before describing the processing performed
22 within the components of the administrative tool framework, exemplary data
23 structures used within the administrative tool framework are described.

24 Exemplary Data Structures for Cmdlet Objects

25

FIGURE 5 is an exemplary data structure for specifying a cmdlet suitable for use within the administrative tool framework shown in FIGURE 2. When completed, the cmdlet may be a management cmdlet, a non-management cmdlet, a host cmdlet, a provider cmdlet, or the like. The following discussion describes the creation of a cmdlet with respect to a system administrator's perspective (i.e., a provider cmdlet). However, each type of cmdlet is created in the same manner and operates in a similar manner. A cmdlet may be written in any language, such as C#. In addition, the cmdlet may be written using a scripting language or the like. When the administrative tool environment operates with the .NET Framework, the cmdlet may be a .NET object.

The provider cmdlet 500 (hereinafter, referred to as cmdlet 500) is a public class having a cmdlet class name (e.g., StopProcess 504). Cmdlet 500 derives from a cmdlet class 506. An exemplary data structure for a cmdlet class 506 is described below in conjunction with FIGURE 6. Each cmdlet 500 is associated with a command attribute 502 that associates a name (e.g., Stop/Process) with the cmdlet 500. The name is registered within the administrative tool environment. As will be described below, the parser looks in the cmdlet registry to identify the cmdlet 500 when a command string having the name (e.g., Stop/Process) is supplied as input on a command line or in a script.

The cmdlet 500 is associated with a grammar mechanism that defines a grammar for expected input parameters to the cmdlet. The grammar mechanism may be directly or indirectly associated with the cmdlet. For example, the cmdlet 500 illustrates a direct grammar association. In this cmdlet 500, one or more public parameters (e.g., ProcessName 510 and PID 512) are declared. The

1 declaration of the public parameters drives the parsing of the input objects to the
2 cmdlet 500. Alternatively, the description of the parameters may appear in an
3 external source, such as an XML document. The description of the parameters in
4 this external source would then drive the parsing of the input objects to the cmdlet.

5 Each public parameter 510, 512 may have one or more attributes (i.e.,
6 directives) associated with it. The directives may be from any of the following
7 categories: parsing directive 521, data validation directive 522, data generation
8 directive 523, processing directive 524, encoding directive 525, and
9 documentation directive 526. The directives may be surrounded by square
10 brackets. Each directive describes an operation to be performed on the following
11 expected input parameter. Some of the directives may also be applied at a class
12 level, such as user-interaction type directives. The directives are stored in the
13 metadata associated with the cmdlet. The application of these attributes is
14 described below in conjunction with FIGURE 12.

15 These attributes may also affect the population of the parameters declared
16 within the cmdlet. One exemplary process for populating these parameters is
17 described below in conjunction with FIGURE 16. The core engine may apply
18 these directives to ensure compliance. The cmdlet 500 includes a first method 530
19 (hereinafter, interchangeably referred to as StartProcessing method 530) and a
20 second method 540 (hereinafter, interchangeably referred to as processRecord
21 method 540). The core engine uses the first and second methods 530, 540 to
22 direct the processing of the cmdlet 500. For example, the first method 530 is
23 executed once and performs set-up functions. The code 542 within the second
24 method 540 is executed for each object (e.g., record) that needs to be processed by
25

1 the cmdlet 500. The cmdlet 500 may also include a third method (not shown) that
2 cleans up after the cmdlet 500.

3 Thus, as shown in FIGURE 5, code 542 within the second method 540 is
4 typically quite brief and does not contain functionality required in traditional
5 administrative tool environments, such as parsing code, data validation code, and
6 the like. Thus, system administrators can develop complex administrative tasks
7 without learning a complex programming language.

8 FIGURE 6 is an exemplary data structure 600 for specifying a cmdlet base
9 class 602 from which the cmdlet shown in FIGURE 5 is derived. The cmdlet base
10 class 602 includes instructions that provide additional functionality whenever the
11 cmdlet includes a hook statement and a corresponding switch is input on the
12 command line or in the script (jointly referred to as command input).

13 The exemplary data structure 600 includes parameters, such as Boolean
14 parameter verbose 610, whatif 620, and confirm 630. As will be explained below,
15 these parameters correspond to strings that may be entered on the command input.
16 The exemplary data structure 600 may also include a security method 640 that
17 determines whether the task being requested for execution is allowed.

18 FIGURE 7 is another exemplary data structure 700 for specifying a cmdlet.
19 In overview, the data structure 700 provides a means for clearly expressing a
20 contract between the administrative tool framework and the cmdlet. Similar to
21 data structure 500, data structure 700 is a public class that derives from a cmdlet
22 class 704. The software developer specifies a cmdletDeclaration 702 that
23 associates a noun/verb pair, such as "get/process" and "format/table", with the
24 cmdlet 700. The noun/verb pair is registered within the administrative tool
25

1 environment. The verb or the noun may be implicit in the cmdlet name. Also,
2 similar to data structure 500, data structure 700 may include one or more public
3 members (e.g., Name 730, Recurse 732), which may be associated with the one or
4 more directives 520-526 described in conjunction with data structure 500.

5 However, in this exemplary data structure 700, each of the expected input
6 parameters 730 and 732 is associated with an input attribute 731 and 733,
7 respectively. The input attributes 731 and 733 specifying that the data for its
8 respective parameter 730 and 732 should be obtained from the command line.
9 Thus, in this exemplary data structure 700, there are not any expected input
10 parameters that are populated from a pipelined object that has been emitted by
11 another cmdlet. Thus, data structure 700 does not override the first method (e.g.,
12 StartProcessing) or the second method (e.g., ProcessRecord) which are provided
13 by the cmdlet base class.

14 The data structure 700 may also include a private member 740 that is not
15 recognized as an input parameter. The private member 740 may be used for
16 storing data that is generated based on one of the directives.

17 Thus, as illustrated in data structure 700, through the use of declaring
18 public properties and directives within a specific cmdlet class, cmdlet developers
19 can easily specify a grammar for the expected input parameters to their cmdlets
20 and specify processing that should be performed on the expected input parameters
21 without requiring the cmdlet developers to generate any of the underlying logic.
22 Data structure 700 illustrates a direct association between the cmdlet and the
23 grammar mechanism. As mentioned above, this associated may also be indirect,
24
25

1 such as by specifying the expected parameter definitions within an external source,
2 such as an XML document.

3 The exemplary process flows within the administrative tool environment
4 are now described.

5 Exemplary Host Processing Flow

6 FIGURE 8 is a logical flow diagram illustrating an exemplary process for
7 host processing that is performed within the administrative tool framework shown
8 in FIGURE 2. The process 800 begins at block 801, where a request has been
9 received to initiate the administrative tool environment for a specific application.
10 The request may have been sent locally through keyboard input, such as selecting
11 an application icon, or remotely through the web services interface of a different
12 computing device. For either scenario, processing continues to block 802.

13 At block 802, the specific application (e.g., host program) on the “target”
14 computing device sets up its environment. This includes determining which
15 subsets of cmdlets (e.g., management cmdlets 232, non-management cmdlets 234,
16 and host cmdlets 218) are made available to the user. Typically, the host program
17 will make all the non-management cmdlets 234 available and its own host cmdlets
18 218 available. In addition, the host program will make a subset of the
19 management cmdlets 234 available, such as cmdlets dealing with processes, disk,
20 and the like. Thus, once the host program makes the subsets of cmdlets available,
21 the administrative tool framework is effectively embedded within the
22 corresponding application. Processing continues to block 804.

23 At block 804, input is obtained through the specific application. As
24 mentioned above, input may take several forms, such as command lines, scripts,
25

1 voice, GUI, and the like. For example, when input is obtained via a command
2 line, the input is retrieve from the keystrokes entered on a keyboard. For a GUI
3 host, a string is composed based on the GUI. Processing continues at block 806.

4 At block 806, the input is provided to other components within the
5 administrative tool framework for processing. The host program may forward the
6 input directly to the other components, such as the parser. Alternatively, the host
7 program may forward the input via one of its host cmdlets. The host cmdlet may
8 convert its specific type of input (e.g., voice) into a type of input (e.g., text string,
9 script) that is recognized by the administrative tool framework. For example,
10 voice input may be converted to a script or command line string depending on the
11 content of the voice input. Because each host program is responsible for
12 converting their type of input to an input recognized by the administrative tool
13 framework, the administrative tool framework can accept input from any number
14 of various host components. In addition, the administrative tool framework
15 provides a rich set of utilities that perform conversions between data types when
16 the input is forwarded via one of its cmdlets. Processing performed on the input
17 by the other components is described below in conjunction with several other
18 figures. Host processing continues at decision block 808.

19 At decision block 808, a determination is made whether a request was
20 received for additional input. This may occur if one of the other components
21 responsible for processing the input needs additional information from the user in
22 order to complete its processing. For example, a password may be required to
23 access certain data, confirmation of specific actions may be needed, and the like.
24 For certain types of host programs (e.g., voice mail), a request such as this may
25 not be appropriate. Thus, instead of querying the user for additional information,

1 the host program may serialize the state, suspend the state, and send a notification
2 so that at a later time the state may be resumed and the execution of the input be
3 continued. In another variation, the host program may provide a default value
4 after a predetermined time period. If a request for additional input is received,
5 processing loops back to block 804, where the additional input is obtained.
6 Processing then continues through blocks 806 and 808 as described above. If no
7 request for additional input is received and the input has been processed,
8 processing continues to block 810.

9 At block 810, results are received from other components within the
10 administrative tool framework. The results may include error messages, status,
11 and the like. The results are in an object form, which is recognized and processed
12 by the host cmdlet within the administrative tool framework. As will be described
13 below, the code written for each host cmdlet is very minimal. Thus, a rich set of
14 output may be displayed without requiring a huge investment in development
15 costs. Processing continues at block 812.

16 At block 812, the results may be viewed. The host cmdlet converts the
17 results to the display style supported by the host program. For example, a returned
18 object may be displayed by a GUI host program using a graphical depiction, such
19 as an icon, barking dog, and the like. The host cmdlet provides a default format
20 and output for the data. The default format and output may utilize the exemplary
21 output processing cmdlets described below in conjunction with FIGURES 19-23.
22 After the results are optionally displayed, the host processing is complete.

23 Exemplary Process Flows for Handling Input

24 FIGURE 9 is a logical flow diagram illustrating an exemplary process for
25 handling input that is performed within the administrative tool framework shown

1 in FIGURE 2. Processing begins at block 901 where input has been entered via a
2 host program and forwarded to other components within the administrative tool
3 framework. Processing continues at block 902.

4 At block 902, the input is received from the host program. In one
5 exemplary administrative tool framework, the input is received by the parser,
6 which deciphers the input and directs the input for further processing. Processing
7 continues at decision block 904.

8 At decision block 904, a determination is made whether the input is a
9 script. The input may take the form of a script or a string representing a command
10 line (hereinafter, referred to as a "command string"). The command string may
11 represent one or more cmdlets pipelined together. Even though the administrative
12 tool framework supports several different hosts, each host provides the input as
13 either a script or a command string for processing. As will be shown below, the
14 interaction between scripts and command strings is recursive in nature. For
15 example, a script may have a line that invokes a cmdlet. The cmdlet itself may be
16 a script.

17 Thus, at decision block 904, if the input is in a form of a script, processing
18 continues at block 906, where processing of the script is performed. Otherwise,
19 processing continues at block 908, where processing of the command string is
20 performed. Once the processing performed within either block 906 or 908 is
21 completed, processing of the input is complete.

22 Exemplary Processing of Scripts

23 FIGURE 10 is a logical flow diagram illustrating a process for processing a
24 script suitable for use within the process for handling input shown in FIGURE 9.
25 The process begins at block 1001, where the input has been identified as a script.

1 The script engine and parser communicate with each other to perform the
2 following functions. Processing continues at block 1002.

3 At block 1002, pre-processing is performed on the script. Briefly, turning
4 to FIGURE 11, a logical flow diagram is shown that illustrates a script pre-
5 processing process 1100 suitable for use within the script processing process 1000.
6 Script pre-processing begins at block 1101 and continues to decision block 1102.

7 At decision block 1102, a determination is made whether the script is being
8 run for the first time. This determination may be based on information obtained
9 from a registry or other storage mechanism. The script is identified from within
10 the storage mechanism and the associated data is reviewed. If the script has not
11 run previously, processing continues at block 1104.

12 At block 1104, the script is registered in the registry. This allows
13 information about the script to be stored for later access by components within the
14 administrative tool framework. Processing continues at block 1106.

15 At block 1106, help and documentation are extracted from the script and
16 stored in the registry. Again, this information may be later accessed by
17 components within the administrative tool framework. The script is now ready for
18 processing and returns to block 1004 in FIGURE 10.

19 Returning to decision block 1102, if the process concludes that the script
20 has run previously, processing continues to decision block 1108. At decision block
21 1108, a determination is made whether the script failed during processing. This
22 information may be obtained from the registry. If the script has not failed, the
23 script is ready for processing and returns to block 1004 in FIGURE 10.

24 However, if the script has failed, processing continues at block 1110. At
25 block 1110, the script engine may notify the user through the host program that the

1 script has previously failed. This notification will allow a user to decide whether
2 to proceed with the script or to exit the script. As mentioned above in conjunction
3 with FIGURE 8, the host program may handle this request in various ways
4 depending on the style of input (e.g., voice, command line). Once additional input
5 is received from the user, the script either returns to block 1004 in FIGURE 10 for
6 processing or the script is aborted.

7 Returning to block 1004 in FIGURE 10, a line from the script is retrieved.
8 Processing continues at decision block 1006. At decision block 1006, a
9 determination is made whether the line includes any constraints. A constraint is
10 detected by a predefined begin character (e.g., a bracket “[”) and a corresponding
11 end character (e.g., a close bracket “]”). If the line includes constraints, processing
12 continues to block 1008.

13 At block 1008, the constraints included in the line are applied. In general,
14 the constraints provide a mechanism within the administrative tool framework to
15 specify a type for a parameter entered in the script and to specify validation logic
16 which should be performed on the parameter. The constraints are not only
17 applicable to parameters, but are also applicable to any type of construct entered in
18 the script, such as variables. Thus, the constraints provide a mechanism within an
19 interpretive environment to specify a data type and to validate parameters. In
20 traditional environments, system administrators are unable to formally test
21 parameters entered within a script. An exemplary process for applying constraints
22 is illustrated in FIGURE 12.

23 At decision block 1010, a determination is made whether the line from the
24 script includes built-in capabilities. Built-in capabilities are capabilities that are
25 not performed by the core engine. Built-in capabilities may be processed using

1 cmdlets or may be processed using other mechanisms, such as in-line functions. If
2 the line does not have built-in capabilities, processing continues at decision block
3 1014. Otherwise, processing continues at block 1012.

4 At block 1012, the built-in capabilities provided on the line of the script are
5 processed. Example built-in capabilities may include execution of control
6 structures, such as “if” statements, “for” loops, switches, and the like. Built-in
7 capabilities may also include assignment type statements (e.g., a=3). Once the
8 built-in capabilities have been processed, processing continues to decision block
9 1014.

10 At decision block 1014, a determination is made whether the line of the
11 script includes a command string. The determination is based on whether the data
12 on the line is associated with a command string that has been registered and with a
13 syntax of the potential cmdlet invocation. As mentioned above, the processing of
14 command strings and scripts may be recursive in nature because scripts may
15 include command strings and command strings may execute a cmdlet that is a
16 script itself. If the line does not include a command string, processing continues at
17 decision block 1018. Otherwise, processing continues at block 1016.

18 At block 1016, the command string is processed. In overview, the
19 processing of the command string includes identifying a cmdlet class by the parser
20 and passing the corresponding cmdlet object to the core engine for execution. The
21 command string may also include a pipelined command string that is parsed into
22 several individual cmdlet objects and individually processed by the core engine.
23 One exemplary process for processing command strings is described below in
24 conjunction with FIGURE 14. Once the command string is processed, processing
25 continues at decision block 1018.

1 At decision block 1018, a determination is made whether there is another
2 line in the script. If there is another line in the script, processing loops back to
3 block 1004 and proceeds as described above in blocks 1004-1016. Otherwise,
4 processing is complete.

5 An exemplary process for applying constraints in block 1008 is illustrated
6 in FIGURE 12. The process begins at block 1201 where a constraint is detected in
7 the script or in the command string on the command line. When the constraint is
8 within a script, the constraints and the associated construct may occur on the same
9 line or on separate lines. When the constraint is within a command string, the
10 constraint and the associated construct occur before the end of line indicator (e.g.,
11 enter key). Processing continues to block 1202.

12 At block 1202, constraints are obtained from the interpretive environment.
13 In one exemplary administrative tool environment, the parser deciphers the input
14 and determines the occurrence of constraints. Constraints may be from one of the
15 following categories: predicate directive, parsing directive, data validation
16 directive, data generation directive, processing directive, encoding directive, and
17 documentation directive. In one exemplary parsing syntax, the directives are
18 surrounded by square brackets and describe the construct that follows them. The
19 construct may be a function, a variable, a script, or the like.

20 As will be described below, through the use of directives, script authors are
21 allowed to easily type and perform processing on the parameters within the script
22 or command line (i.e., an interpretive environment) without requiring the script
23 authors to generate any of the underlying logic. Processing continues to block
24 1204.
25

1 At block 1204, the constraints that are obtained are stored in the metadata
2 for the associated construct. The associated construct is identified as being the
3 first non-attribution token after one or more attribution tokens (tokens that denote
4 constraints) have been encountered. Processing continues to block 1206.

5 At block 1206, whenever the construct is encountered within the script or in
6 the command string, the constraints defined within the metadata are applied to the
7 construct. The constraints may include data type, predicate directives 1210,
8 documentation directives 1212, parsing directives 1214, data generation directives
9 1216, data validation directives 1218, and object processing and encoding
10 directives 1220. Constraints specifying data types may specify any data type
11 supported by the system on which the administrative tool framework is running.
12 Predicate directives 1210 are directives that indicate whether processing should
13 occur. Thus, predicate directives 1210 ensure that the environment is correct for
14 execution. For example, a script may include the following predicate directive:

15 [PredicateScript("isInstalled","ApplicationZ")].

16 The predicate directive ensures that the correct application is installed on
17 the computing device before running the script. Typically, system environment
18 variables may be specified as predicate directives. Exemplary directives from
19 directive types 1212-1220 are illustrated in Tables 1-5. Processing of the script is
20 then complete.

21 Thus, the present process for applying types and constraints within an
22 interpretive environment, allows system administrators to easily specify a type,
23 specify validation requirements, and the like without having to write the
24
25

underlying logic for performing this processing. The following is an example of the constraint processing performed on a command string specified as follows:

[Integer][ValidationRange(3,5)]\$a=4.

There are two constraints specified via attribution tokens denoted by “[]”. The first attribution token indicates that the variable is a type integer and a second attribution token indicates that the value of the variable \$a must be between 3 and 5 inclusive. The example command string ensures that if the variable \$a is assigned in a subsequent command string or line, the variable \$a will be checked against the two constraints. Thus, the following command strings would each result in an error:

\$a = 231

\$a = “apple”

\$a = \$(get/location).

The constraints are applied at various stages within the administrative tool framework. For example, applicability directives, documentation directives, and parsing guideline directives are processed at a very early stage within the parser. Data generation directives and validation directives are processed in the engine once the parser has finished parsing all the input parameters.

The following tables illustrate representative directives for the various categories, along with an explanation of the processing performed by the administrative tool environment in response to the directive.

Name	Description
------	-------------

PrerequisiteMachineRoleAttribute	Informs shell whether element is to be used only in certain machine roles (e.g., File Server, Mail Server).
PrerequisiteUserRoleAttribute	Informs shell whether element is to be used only in certain user roles (e.g., Domain Administrator, Backup Operator).
PrerequisiteScriptAttribute	Informs the shell this script will be run before excuting the actual command or parameter. Can be used for parameter validation
PrerequisiteUITypeAttribute	This is used to check the User interface available before excuting

Table 1. Applicability Directives

Name	Description
ParsingParameterPositionAttribute	Maps unqualified parameters based on position.
ParsingVariableLengthParameterListAttribute	Maps parameters not having a Parsing ParameterPosition

1		attribute.
2	ParsingDisallowInteractionAttribute	Specifies action
3		when number of
4		parameters is less than
5		required number.
6	ParsingRequireInteractionAttribute	Specifies that
7		parameters are obtained
8		through interaction.
9	ParsingHiddenElementAttribute	Makes parameter
10		invisible to end user.
11	ParsingMandatoryParameterAttribute	Specifies that the
12		parameter is required.
13	ParsingPasswordParameterAttribute	Requires special
14		handling of parameter.
15	ParsingPromptStringAttribute	Specifies a prompt
16		for the parameter.
17	ParsingDefaultAnswerAttribute	Specifies default
18		answer for parameter.
19	ParsingDefaultAnswerScriptAttribute	Specifies action to
20		get default answer for
21		parameter.
22	ParsingDefaultValueAttribute	Specifies default
23		value for parameter.
24	ParsingDefaultValueScriptAttribute	Specifies action to
25		

	get default value for parameter.
ParsingParameterMappingAttribute	Specifies a way to group parameters
ParsingParameterDeclarationAttribute	This defines that the filed is a parameter
ParsingAllowPipelineInputAttribute	Defines the parameter can be populated from the pipeline

Table 2. Parsing Guideline Directives

Name	Description
DocumentNameAttribute	Provides a Name to refer to elements for interaction or help.
DocumentShortDescriptionAttribute	Provides brief description of element.
DocumentLongDescriptionAttribute	Provides detailed description of element.
DocumentExampleAttribute	Provides example of element.
DocumentSeeAlsoAttribute	Provides a list of related elements.
DocumentSynopsisAttribute	Provides documentation

information for element.

Table 3. Documentation Directives

Name	Description
ValidationRangeAttribute	Specifies that parameter must be within certain range.
ValidationSetAttribute	Specifies that parameter must be within certain collection.
ValidationPatternAttribute	Specifies that parameter must fit a certain pattern.
ValidationLengthAttribute	Specifies the strings must be within size range.
ValidationTypeAttribute	Specifies that parameter must be of certain type.
ValidationCountAttribute	Specifies that input items must be of a certain number.
ValidationFileAttribute	Specifies certain properties for a file.
ValidationFileAttributesAttribute	Specifies certain properties for a file.
ValidationFileSizeAttribute	Specifies that files must be within specified range.

ValidationNetworkAttribute	Specifies that given Network Entity supports certain properties.
ValidationScriptAttribute	Specifies conditions to evaluate before using element.
ValidationMethodAttribute	Specifies conditions to evaluate before using element.

Table 4. Data Validation Directives

Name	Description
ProcessingTrimStringAttribute	Specifies size limit for strings.
ProcessingTrimCollectionAttribute	Specifies size limit for collection.
EncodingTypeCoercionAttribute	Specifies Type that objects are to be encoded.
ExpansionWildcardsAttribute	Provides a mechanism to allow globbing

Table 5. Processing and Encoding Directives

When the exemplary administrative tool framework is operating within the .NET™ Framework, each category has a base class that is derived from a basic category class (e.g., CmdAttribute). The basic category class derives from a

1 System.Attribute class. Each category has a pre-defined function (e.g.,
2 attrib.func()) that is called by the parser during category processing. The script
3 author may create a custom category that is derived from a custom category class
4 (e.g., CmdCustomAttribute). The script author may also extend an existing
5 category class by deriving a directive class from the base category class for that
6 category and override the pre-defined function with their implementation. The
7 script author may also override directives and add new directives to the pre-
8 defined set of directives.

9 The order of processing of these directives may be stored in an external
10 data store accessible by the parser. The administrative tool framework looks for
11 registered categories and calls a function (e.g., ProcessCustomDirective) for each
12 of the directives in that category. Thus, the order of category processing may be
13 dynamic by storing the category execution information in a persistent store. At
14 different processing stages, the parser checks in the persistent store to determine if
15 any metadata category needs to be executed at that time. This allows categories to
16 be easily deprecated by removing the category entry from the persistent store.

17 Exemplary Processing of Command Strings

18 One exemplary process for processing command strings is now described.
19 FIGURE 13 is a functional flow diagram graphically illustrating the processing of
20 a command string 1350 through a parser 220 and a core engine 224 within the
21 administrative tool framework shown in FIGURE 2. The exemplary command
22 string 1350 pipelines several commands (i.e., process command 1360, where
23 command 1362, sort command 1364, and table command 1366). The command
24 line 1350 may pass input parameters to any of the commands (e.g., "handlecount >
25

1 400" is passed to the where command 1362). One will note that the process
2 command 1360 does not have any associated input parameters.

3 In the past, each command was responsible for parsing the input parameters
4 associated with the command, determining whether the input parameters were
5 valid, and issuing error messages if the input parameters were not valid. Because
6 the commands were typically written by various programmers, the syntax for the
7 input parameters on the command line was not very consistent. In addition, if an
8 error occurred, the error message, even for the same error, was not very consistent
9 between the commands.

10 For example, in a UNIX environment, an "ls" command and a "ps"
11 command have many inconsistencies between them. While both accept an option
12 "-w", the "-w" option is used by the "ls" command to denote the width of the page,
13 while the "-w" option is used by the "ps" command to denote print wide output (in
14 essence, ignoring page width). The help pages associated with the "ls" and the
15 "ps" command have several inconsistencies too, such as having options bolded in
16 one and not the other, sorting options alphabetically in one and not the other,
17 requiring some options to have dashes and some not.

18 The present administrative tool framework provides a more consistent
19 approach and minimizes the amount of duplicative code that each developer must
20 write. The administrative tool framework 200 provides a syntax (e.g., grammar), a
21 corresponding semantics (e.g., a dictionary), and a reference model to enable
22 developers to easily take advantage of common functionality provided by the
23 administrative tool framework 200.
24
25

1 Before describing the present invention any further, definitions for
2 additional terms appearing through-out this specification are provided. Input
3 parameter refers to input-fields for a cmdlet. Argument refers to an input
4 parameter passed to a cmdlet that is the equivalent of a single string in the argv
5 array or passed as a single element in a cmdlet object. As will be described below,
6 a cmdlet provides a mechanism for specifying a grammar. The mechanism may
7 be provided directly or indirectly. An argument is one of an option, an option-
8 argument, or an operand following the command-name. Examples of arguments
9 are given based on the following command line:

10
11 findstr /i /d:\winnt;\winnt\system32 aa*b *.ini.
12
13

14 In the above command line, "findstr" is argument 0, "/i" is argument 1,
15 "/d:\winnt;\winnt\system32" is argument 2, "aa*b" is argument 3, and "*.ini" is
16 argument 4. An "option" is an argument to a cmdlet that is generally used to
17 specify changes to the program's default behavior. Continuing with the example
18 command line above, "/i" and "/d" are options. An "option-argument" is an input
19 parameter that follows certain options. In some cases, an option-argument is
20 included within the same argument string as the option. In other cases, the option-
21 argument is included as the next argument. Referring again to the above
22 command line, "winnt;\winnt\system32" is an option-argument. An "operand" is
23 an argument to a cmdlet that is generally used as an object supplying information
24 to a program necessary to complete program processing. Operands generally
25 follow the options in a command line. Referring to the example command line

1 above again, "aa*b" and "*.ini" are operands. A "parsable stream" includes the
2 arguments.

3 Referring to FIGURE 13, parser 220 parses a parsable stream (e.g.,
4 command string 1350) into constituent parts 1320-1326 (e.g., where portion 1322).
5 Each portion 1320-1326 is associated with one of the cmdlets 1330-1336. Parser
6 220 and engine 224 perform various processing, such as parsing, parameter
7 validation, data generation, parameter processing, parameter encoding, and
8 parameter documentation. Because parser 220 and engine 224 perform common
9 functionality on the input parameters on the command line, the administrative tool
10 framework 200 is able to issue consistent error messages to users.

11 As one will recognize, the executable cmdlets 1330-1336 written in
12 accordance with the present administrative tool framework require less code than
13 commands in prior administrative environments. Each executable cmdlet 1330-
14 1336 is identified using its respective constituent part 1320-1326. In addition,
15 each executable cmdlet 1330-1336 outputs objects (represented by arrows 1340,
16 1342, 1344, and 1346) which are input as input objects (represented by arrows
17 1341, 1343, and 1345) to the next pipelined cmdlet. These objects may be input
18 by passing a reference (e.g., handle) to the object. The executable cmdlets 1330-
19 1336 may then perform additional processing on the objects that were passed in.

20
21 FIGURE 14 is a logical flow diagram illustrating in more detail the
22 processing of command strings suitable for use within the process for handling
23 input shown in FIGURE 9. The command string processing begins at block 1401,
24 where either the parser or the script engine identified a command string within the
25 input. In general the core engine performs set-up and sequencing of the data flow

1 of the cmdlets. The set-up and sequencing for one cmdlet is described below, but
2 is applicable to each cmdlet in a pipeline. Processing continues at block 1404.

3 At block 1404, a cmdlet is identified. The identification of the cmdlet may
4 be thru registration. The core engine determines whether the cmdlet is local or
5 remote. The cmdlet may execute in the following locations: 1) within the
6 application domain of the administrative tool framework; 2) within another
7 application domain of the same process as the administrative tool framework; 3)
8 within another process on the same computing device; or 4) within a remote
9 computing device. The communication between cmdlets operating within the
10 same process is through objects. The communication between cmdlets operating
11 within different processes is through a serialized structured data format. One
12 exemplary serialized structured data format is based on the extensible markup
13 language (XML). Processing continues at block 1406.

14 At block 1406, an instance of the cmdlet object is created. An exemplary
15 process for creating an instance of the cmdlet is described below in conjunction
16 with FIGURE 15. Once the cmdlet object is created, processing continues at
17 block 1408.

18 At block 1408, the properties associated with the cmdlet object are
19 populated. As described above, the developer declares properties within a cmdlet
20 class or within an external source. Briefly, the administrative tool framework will
21 decipher the incoming object(s) to the cmdlet instantiated from the cmdlet class
22 based on the name and type that is declared for the property. If the types are
23 different, the type may be coerced via the extended data type manager. As
24 mentioned earlier, in pipelined command strings, the output of each cmdlet may be
25 a list of handles to objects. The next cmdlet may inputs this list of object handles,

1 performs processing, and passes another list of object handles to the next cmdlet.
2 In addition, as illustrated in FIGURE 7, input parameters may be specified as
3 coming from the command line. One exemplary method for populating properties
4 associated with a cmdlet is described below in conjunction with FIGURE 16.
5 Once the cmdlet is populated, processing continues at block 1410.

6 At block 1410, the cmdlet is executed. In overview, the processing
7 provided by the cmdlet is performed at least once, which includes processing for
8 each input object to the cmdlet. Thus, if the cmdlet is the first cmdlet within a
9 pipelined command string, the processing is executed once. For subsequent
10 cmdlets, the processing is executed for each object that is passed to the cmdlet.
11 One exemplary method for executing cmdlets is described below in conjunction
12 with FIGURE 5. When the input parameters are only coming from the command
13 line, execution of the cmdlet uses the default methods provided by the base cmdlet
14 case. Once the cmdlet is finished executing, processing proceeds to block 1412.

15 At block 1412, the cmdlet is cleaned-up. This includes calling the
16 destructor for the associated cmdlet object which is responsible for de-allocating
17 memory and the like. The processing of the command string is then complete.

18 Exemplary Process for Creating a Cmdlet Object

19 FIGURE 15 is a logical flow diagram illustrating an exemplary process for
20 creating a cmdlet object suitable for use within the processing of command strings
21 shown in FIGURE 14. At this point, the cmdlet data structure has been developed
22 and attributes and expected input parameters have been specified. The cmdlet has
23 been compiled and has been registered. During registration, the class name (i.e.,
24 cmdlet name) is written in the registration store and the metadata associated with
25 the cmdlet has been stored. The process 1500 begins at block 1501, where the

1 parser has received input (e.g., keystrokes) indicating a cmdlet. The parser may
2 recognize the input as a cmdlet by looking up the input from within the registry
3 and associating the input with one of the registered cmdlets. Processing proceeds
4 to block 1504.

5 At block 1504, metadata associated with the cmdlet object class is read.
6 The metadata includes any of the directives associated with the cmdlet. The
7 directives may apply to the cmdlet itself or to one or more of the parameters.
8 During cmdlet registration, the registration code registers the metadata into a
9 persistent store. The metadata may be stored in an XML file in a serialized
10 format, an external database, and the like. Similar to the processing of directives
11 during script processing, each category of directives is processed at a different
12 stage. Each metadata directive handles its own error handling. Processing
13 continues at block 1506.

14 At block 1506, a cmdlet object is instantiated based on the identified cmdlet
15 class. Processing continues at block 1508.

16 At block 1508, information is obtained about the cmdlet. This may occur
17 through reflection or other means. The information is about the expected input
18 parameters. As mentioned above, the parameters that are declared public (e.g.,
19 public string Name 730) correspond to expected input parameters that can be
20 specified in a command string on a command line or provided in an input stream.
21 The administrative tool framework through the extended type manager, described
22 in FIGURE 18, provides a common interface for returning the information (on a
23 need basis) to the caller. Processing continues at block 1510.
24
25

1 At block 1510, applicability directives (e.g., Table 1) are applied. The
2 applicability directives insure that the class is used in certain machine roles and/or
3 user roles. For example, certain cmdlets may only be used by Domain
4 Administrators. If the constraint specified in one of the applicability directives is
5 not met, an error occurs. Processing continues at block 1512.

6 At block 1512, metadata is used to provide intellisense. At this point in
7 processing, the entire command string has not yet been entered. The
8 administrative tool framework, however, knows the available cmdlets. Once a
9 cmdlet has been determined, the administrative tool framework knows the input
10 parameters that are allowed by reflecting on the cmdlet object. Thus, the
11 administrative tool framework may auto-complete the cmdlet once a
12 disambiguating portion of the cmdlet name is provided, and then auto-complete
13 the input parameter once a disambiguating portion of the input parameter has been
14 typed on the command line. Auto-completion may occur as soon as the portion of
15 the input parameter can identify one of the input parameters unambiguously. In
16 addition, auto-completion may occur on cmdlet names and operands too.
17 Processing continues at block 1514.

18 At block 1514, the process waits until the input parameters for the cmdlet
19 have been entered. This may occur once the user has indicated the end of the
20 command string, such as by hitting a return key. In a script, a new line indicates
21 the end of the command string. This wait may include obtaining additional
22 information from the user regarding the parameters and applying other directives.
23 When the cmdlet is one of the pipelined parameters, processing may begin
24
25

1 immediately. Once, the necessary command string and input parameters have
2 been provided, processing is complete.

3 Exemplary Process for Populating the Cmdlet

4 An exemplary process for populating a cmdlet is illustrated in FIGURE 16
5 and is now described, in conjunction with FIGURE 5. In one exemplary
6 administrative tool framework, the core engine performs the processing to
7 populate the parameters for the cmdlet. Processing begins at block 1601 after an
8 instance of a cmdlet has been created. Processing continues to block 1602.

9 At block 1602, a parameter (e.g., ProcessName) declared within the cmdlet
10 is retrieved. Based on the declaration with the cmdlet, the core engine recognizes
11 that the incoming input objects will provide a property named "ProcessName". If
12 the type of the incoming property is different than the type specified in the
13 parameter declaration, the type will be coerced via the extended type manager.
14 The process of coercing data types is explained below in the subsection entitled
15 "Exemplary Extended Type Manager Processing." Processing continues to block
16 1603.

17 At block 1603, an attribute associated with the parameter is obtained. The
18 attribute identifies whether the input source for the parameter is the command line
19 or whether it is from the pipeline. Processing continues to decision block 1604.

20 At decision block 1604, a determination is made whether the attribute
21 specifies the input source as the command line. If the input source is the
22 command line, processing continues at block 1609. Otherwise, processing
23 continues at decision block 1605.
24
25

1 At decision block 1605, a determination is made whether the property name
2 specified in the declaration should be used or whether a mapping for the property
3 name should be used. This determination is based on whether the command input
4 specified a mapping for the parameter. The following line illustrates an exemplary
5 mapping of the parameter "ProcessName" to the "foo" member of the incoming
6 object:

7 \$ get/process | where han* -gt 500 | stop/process -ProcessName<-foo.

8 Processing continues at block 1606.

9 At block 1606, the mapping is applied. The mapping replaces the name of
10 the expected parameter from "ProcessName" to "foo", which is then used by the
11 core engine to parse the incoming objects and to identify the correct expected
12 parameter. Processing continues at block 1608.

13 At block 1608, the extended type manager is queried to locate a value for
14 the parameter within the incoming object. As explain in conjunction with the
15 extended type manager, the extended type manager takes the parameter name and
16 uses reflection to identify a parameter within the incoming object with parameter
17 name. The extended type manager may also perform other processing for the
18 parameter, if necessary. For example, the extended type manager may coerce the
19 type of data to the expected type of data through a conversion mechanism
20 described above. Processing continues to decision block 1610.

21 Referring back to block 1609, if the attribute specifies that the input source
22 is the command line, data from the command line is obtained. Obtaining the data
23 from the command line may be performed via the extended type manager.
24 Processing then continues to decision block 1610.

1 At decision block 1610, a determination is made whether there is another
2 expected parameter. If there is another expected parameter, processing loops back
3 to block 1602 and proceeds as described above. Otherwise, processing is
4 complete and returns.

5 Thus, as shown, cmdlets act as a template for shredding incoming data to
6 obtain the expected parameters. In addition, the expected parameters are obtained
7 without knowing the type of incoming object providing the value for the expected
8 parameter. This is quite different than traditional administrative environments.
9 Traditional administrative environments are tightly bound and require that the type
10 of object be known at compile time. In addition, in traditional environments, the
11 expected parameter would have been passed into the function by value or by
12 reference. Thus, the present parsing (e.g., "shredding") mechanism allows
13 programmers to specify the type of parameter without requiring them to
14 specifically know how the values for these parameters are obtained.

15 For example, given the following declaration for the cmdlet Foo:

```
16  
17  
18       class Foo : Cmdlet  
19       {  
20               string Name;  
21               Bool Recurse;  
22       }  
23  
24
```

25 The command line syntax may be any of the following:

1
2 \$ Foo -Name: (string) -Recurse: True

3 \$ Foo -Name <string> -Recurse True

4 \$Foo -Name (string).
5

6
7 The set of rules may be modified by system administrators in order to yield
8 a desired syntax. In addition, the parser may support multiple sets of rules, so that
9 more than one syntax can be used by users. In essence, the grammar associated
10 with the cmdlet structure (e.g., string Name and Bool Recurse) drives the parser.

11 In general, the parsing directives describe how the parameters entered as
12 the command string should map to the expected parameters identified in the
13 cmdlet object. The input parameter types are checked to determine whether
14 correct. If the input parameter types are not correct, the input parameters may be
15 coerced to become correct. If the input parameter types are not correct and can not
16 be coerced, a usage error is printed. The usage error allows the user to become
17 aware of the correct syntax that is expected. The usage error may obtain
18 information describing the syntax from the Documentation Directives. Once the
19 input parameter types have either been mapped or have been verified, the
20 corresponding members in the cmdlet object instance are populated. As the
21 members are populated, the extended type manager provides processing of the
22 input parameter types. Briefly, the processing may include a property path
23 mechanism, a key mechanism, a compare mechanism, a conversion mechanism, a
24 globber mechanism, a relationship mechanism, and a property set mechanism.
25

1 Each of these mechanisms is described in detail below in the section entitled
2 “Extended Type Manager Processing”, which also includes illustrative examples.

3 Exemplary Process for Executing the Cmdlet

4 An exemplary process for executing a cmdlet is illustrated in FIGURE 17
5 and is now described. In one exemplary administrative tool environment, the core
6 engine executes the cmdlet. As mentioned above, the code 1442 within the second
7 method 1440 is executed for each input object. Processing begins at block 1701
8 where the cmdlet has already been populated. Processing continues at block 1702.

9 At block 1702, a statement from the code 542 is retrieved for execution.
10 Processing continues at decision block 1704.

11 At decision block 1704, a determination is made whether a hook is included
12 within the statement. Turning briefly to FIGURE 5, the hook may include calling
13 an API provided by the core engine. For example, statement 550 within the code
14 542 of cmdlet 500 in FIGURE 5 calls the confirmprocessing API specifying the
15 necessary parameters, a first string (e.g., “PID=”), and a parameter (e.g., PID).
16 Turning back to FIGURE 17, if the statement includes the hook, processing
17 continues to block 1712. Thus, if the instruction calling the confirmprocessing
18 API is specified, the cmdlet operates in an alternate executing mode that is
19 provided by the operating environment. Otherwise, processing continues at block
20 1706 and execution continues in the “normal” mode.

21 At block 1706, the statement is processed. Processing then proceeds to
22 decision block 1708. At block 1708, a determination is made whether the code
23 includes another statement. If there is another statement, processing loops back to
24
25

1 block 1702 to get the next statement and proceeds as described above. Otherwise,
2 processing continues to decision block 1714.

3 At decision block 1714, a determination is made whether there is another
4 input object to process. If there is another input object, processing continues to
5 block 1716 where the cmdlet is populated with data from the next object. The
6 population process described in FIGURE 16 is performed with the next object.
7 Processing then loops back to block 1702 and proceeds as described above. Once
8 all the objects have been processed, the process for executing the cmdlet is
9 complete and returns.

10 Returning back to decision block 1704, if the statement includes the hook,
11 processing continues to block 1712. At block 1712, the additional features
12 provided by the administrative tool environment are processed. Processing
13 continues at decision block 1708 and continues as described above.

14 The additional processing performed within block 1712 is now described in
15 conjunction with the exemplary data structure 600 illustrated in FIGURE 6. As
16 explained above, within the command base class 600 there may be parameters
17 declared that correspond to additional expected input parameters (e.g., a switch).

18 The switch includes a predetermined string, and when recognized, directs
19 the core engine to provide additional functionality to the cmdlet. If the parameter
20 verbose 610 is specified in the command input, verbose statements 614 are
21 executed. The following is an example of a command line that includes the
22 verbose switch:

23
24 \$ get/process | where "han* -gt 500" | stop/process -verbose.
25

1 In general, when “-verbose” is specified within the command input, the
2 core engine executes the command for each input object and forwards the actual
3 command that was executed for each input object to the host program for display.
4 The following is an example of output generated when the above command line is
5 executed in the exemplary administrative tool environment:

6
7 \$ stop/process PID=15
8 \$ stop/process PID=33.

9
10 If the parameter whatif 620 is specified in the command input, whatif
11 statements 624 are executed. The following is an example of a command line that
12 includes the whatif switch:

13
14 \$ get/process | where “han* -gt 500” | stop/process -whatif.

15
16 In general, when “-whatif” is specified, the core engine does not actually
17 execute the code 542, but rather sends the commands that would have been
18 executed to the host program for display. The following is an example of output
19 generated when the above command line is executed in the administrative tool
20 environment of the present invention:

21
22 # \$ stop/process PID=15
23 # \$ stop/process PID=33.

1 If the parameter confirm 630 is specified in the command input, confirm
2 statements 634 are executed. The following is an example of a command line that
3 includes the confirm switch:

4
5 \$ get/process | where "han* -gt 500" | stop/process -confirm.
6

7 In general, when "-confirm" is specified, the core engine requests
8 additional user input on whether to proceed with the command or not. The
9 following is an example of output generated when the above command line is
10 executed in the administrative tool environment of the present invention.

11
12 \$ stop/process PID 15

13 Y/N Y

14 \$ stop/process PID 33

15 Y/N N.
16

17 As described above, the exemplary data structure 600 may also include a
18 security method 640 that determines whether the task being requested for
19 execution should be allowed. In traditional administrative environments, each
20 command is responsible for checking whether the person executing the command
21 has sufficient privileges to perform the command. In order to perform this check,
22 extensive code is needed to access information from several sources. Because of
23 these complexities, many commands did not perform a security check. The
24 inventors of the present administrative tool environment recognized that when the
25 task is specified in the command input, the necessary information for performing

1 the security check is available within the administrative tool environment.
2 Therefore, the administrative tool framework performs the security check without
3 requiring complex code from the tool developers. The security check may be
4 performed for any cmdlet that defines the hook within its cmdlet. Alternatively,
5 the hook may be an optional input parameter that can be specified in the command
6 input, similar to the verbose parameter described above.

7 The security check is implemented to support roles based authentication,
8 which is generally defined as a system of controlling which users have access to
9 resources based on the role of the user. Thus, each role is assigned certain access
10 rights to different resources. A user is then assigned to one or more roles. In
11 general, roles based authentication focus on three items: principle, resource, and
12 action. The *principle* identifies who requested the *action* to be performed on the
13 *resource*.

14 The inventors of the present invention recognized that the cmdlet being
15 requested corresponded to the action that was to be performed. In addition, the
16 inventors appreciated that the owner of the process in which the administrative
17 tool framework was executing corresponded to the principle. Further, the
18 inventors appreciated that the resource is specified within the cmdlet. Therefore,
19 because the administrative tool framework has access to these items, the inventors
20 recognized that the security check could be performed from within the
21 administrative tool framework without requiring tool developers to implement the
22 security check.

23 The operation of the security check may be performed any time additional
24 functionality is requested within the cmdlet by using the hook, such as the
25 confirmprocessing API. Alternatively, security check may be performed by

1 checking whether a security switch was entered on the command line, similar to
2 verbose, whatif, and confirm. For either implementation, the checkSecurity
3 method calls an API provided by a security process (not shown) that provides a set
4 of APIs for determining who is allowed. The security process takes the
5 information provided by the administrative tool framework and provides a result
6 indicating whether the task may be completed. The administrative tool framework
7 may then provide an error or just stop the execution of the task.

8 Thus, by providing the hook within the cmdlet, the developers may use
9 additional processing provided by the administrative tool framework.

10 Exemplary Extended Type Manager Processing

11 As briefly mentioned above in conjunction with FIGURE 18, the extended
12 type manager may perform additional processing on objects that are supplied. The
13 additional processing may be performed at the request of the parser 220, the script
14 engine 222, or the pipeline processor 402. The additional processing includes a
15 property path mechanism, a key mechanism, a compare mechanism, a conversion
16 mechanism, a globber mechanism, a relationship mechanism, and a property set
17 mechanism. Those skilled in the art will appreciate that the extended type
18 manager may also be extended with other processing without departing from the
19 scope of the claimed invention. Each of the additional processing mechanisms is
20 now described.

21 First, the property path mechanism allows a string to navigate properties of
22 objects. In current reflection systems, queries may query properties of an object.
23 However, in the present extended type manager, a string may be specified that will
24 provide a navigation path to successive properties of objects. The following is an
25 illustrative syntax for the property path: P1.P2.P3.P4.

Each component (e.g., P1, P2, P3, and P4) comprises a string that may represent a property, a method with parameters, a method without parameters, a field, an XPATH, or the like. An XPATH specifies a query string to search for an element (e.g., `"/FOO@=13"`). Within the string, a special character may be included to specifically indicate the type of component. If the string does not contain the special character, the extended type manager may perform a lookup to determine the type of component. For example, if component P1 is an object, the extended type manager may query whether P2 is a property of the object, a method on the object, a field of the object, or a property set. Once the extended type manager identifies the type for P2, processing according to that type is performed. If the component is not one of the above types, the extended type manager may further query the extended sources to determine whether there is a conversion function to convert the type of P1 into the type of P2. These and other lookups will now be described using illustrative command strings and showing the respective output.

The following is an illustrative string that includes a property path:

```
$ get/process | /where hand* -gt> 500 | format/table name.toupper, ws.kb, exe*.ver*.description.tolower.trunc(30).
```

In the above illustrative string, there are three property paths: (1) `"name.toupper"`; (2) `"ws.kb"`; and (3) `"exe*.ver*.description.tolower.trunc(30)"`. Before describing these property paths, one should note that `"name"`, `"ws"`, and `"exe"` specify the properties for the table. In addition, one should note that each of these properties is a direct property of the incoming object, originally generated by

1 “get/process” and then pipelined through the various cmdlets. Processing
2 involved for each of the three property paths will now be described.

3 In the first property path (i.e., “name.toupper”), name is a direct property of
4 the incoming object and is also an object itself. The extended type manager
5 queries the system using the priority lookup described above to determine the type
6 for toupper. The extended type manager discovers that toupper is not a property.
7 However, toupper may be a method inherited by a string type to convert lower
8 case letters to upper case letters within the string. Alternatively, the extended type
9 manager may have queried the extended metadata to determine whether there is
10 any third party code that can convert a name object to upper case. Upon finding
11 the component type, processing is performed in accordance with that component
12 type.

13 In the second property path (i.e., “ws.kb”), “ws” is a direct property of the
14 incoming object and is also an object itself. The extended type manager
15 determines that “ws” is an integer. Then, the extended type manager queries
16 whether kb is a property of an integer, whether kb is a method of an integer, and
17 finally queries whether any code knows how to take an integer and convert the
18 integer to a kb type. Third party code is registered to perform this conversion and
19 the conversion is performed.

20 In the third property path (i.e., “exe*.ver*.description.tolower.trunc(30)”
21 there are several components. The first component (“exe*”) is a direct property of
22 the incoming object and is also an object. Again, the extended type manager
23 proceeds down the lookup query in order to process the second component
24 (“ver*”). The “exe*” object does not have a “ver*” property or method, so the
25

1 extend type manager queries the extended metadata to determine whether there is
2 any code that is registered to convert an executable name into a version. For this
3 example, such code exists. The code may take the executable name string and use
4 it to open a file, then accesses the version block object, and return the description
5 property (the third component ("description") of the version block object. The
6 extended type manager then performs this same lookup mechanism for the fourth
7 component ("tolower") and the fifth component ("trunc(40)"). Thus, as
8 illustrated, the extended type manager may perform quite elaborate processing on
9 a command string without the administrator needing to write any specific code.

10 Table 1 illustrates output generated for the illustrative string.

11
12 Name.toupper ws.kb exe*.ver*.description.tolower.trunc(30)

13 ETCLIENT 29,964 etclient

14 CSRSS 6,944

15 SVCHOST 28,944 generic host process for win32

16 OUTLOOK 18,556 office outlook

17 MSMSGs 13,248 messenger

18 Table 1.

19 Another query mechanism 1824 includes a key. The key identifies one or
20 more properties that make an instance of the data type unique. For example, in a
21 database, one column may be identified as the key which can uniquely identify
22 each row (e.g., social security number). The key is stored within the type
23 metadata 1840 associated with the data type. This key may then be used by the
24 extended type manager when processing objects of that data type. The data type
25 may be an extended data type or an existing data type.

Another query mechanism 1824 includes a compare mechanism. The compare mechanism compares two objects. If the two objects directly support the compare function, the directly supported compare function is executed. However, if neither object supports a compare function, the extended type manager may look in the type metadata for code that has been registered to support the compare between the two objects. An illustrative series of command line strings invoking the compare mechanism is shown below, along with corresponding output in Table 2.

```
$ $a = $( get/date )
$ start/sleep 5
$ $b = $( get/date
compare/time $a $b
```

```
Ticks      : 51196579
Days       : 0
Hours      : 0
Milliseconds : 119
Minutes    : 0
Seconds    : 5
TotalDays   : 5.92552997685185E-05
TotalHours  : 0.00142212719444444
TotalMilliseconds : 5119.6579
TotalMinutes : 0.0853276316666667
TotalSeconds : 5.1196579
```

Table 2.

1 Compare/time cmdlet is written to compare two datetime objects. In this
2 case, the DateTime object supports the IComparable interface.

3
4 Another query mechanism 1824 includes a conversion mechanism. The
5 extended type manager allows code to be registered stating its ability to perform a
6 specific conversion. Then, when an object of type A is input and a cmdlet
7 specifies an object of type B, the extended type manager may perform the
8 conversion using one of the registered conversions. The extended type manager
9 may perform a series of conversions in order to coerce type A into type B. The
10 property path described above ("ws.kb") illustrates a conversion mechanism.

11
12 Another query mechanism 1824 includes a globber mechanism. A globber
13 refers to a wild card character within a string. The globber mechanism inputs the
14 string with the wild card character and produces a set of objects. The extended
15 type manager allows code to be registered that specifies wildcard processing. The
16 property path described above ("exe*.ver*.description.tolower.trunc(30))
17 illustrates the globber mechanism. A registered process may provide globbing for
18 file names, file objects, incoming properties, and the like.

19 Another query mechanism 1824 includes a property set mechanism. The
20 property set mechanism allows a name to be defined for a set of properties. An
21 administrator may then specify the name within the command string to obtain the
22 set of properties. The property set may be defined in various ways. In one way, a
23 predefined parameter, such as "?", may be entered as an input parameter for a
24 cmdlet. The operating environment upon recognizing the predefined parameter
25 lists all the properties of the incoming object. The list may be a GUI that allows

an administrator to easily check (e.g., “click on”) the properties desired and name the property set. The property set information is then stored in the extended metadata. An illustrative string invoking the property set mechanism is shown below, along with corresponding output in Table 3:

```
$ get/process | where han* -gt 500 | format/table config.
```

In this illustrative string, a property set named “config” has been defined to include a name property, a process id property (Pid), and a priority property. The output for the table is shown below.

<u>Name</u>	<u>Pid</u>	<u>Priority</u>
ETClient	3528	Normal
csrss	528	Normal
svchost	848	Normal
OUTLOOK	2,772	Normal
msmsgs	2,584	Normal

Table 3.

Another query mechanism 1824 includes a relationship mechanism. In contrast to traditional type systems that support one relationship (i.e., inheritance), the relationship mechanism supports expressing more than one relationship between types. Again, these relationships are registered. The relationship may include finding items that the object consumes or finding the items that consume the object. The extended type manager may access ontologies that describe various relationships. Using the extended metadata and the code, a specification for accessing any ontology service, such as OWL, DAWL, and the like, may be

1 described. The following is a portion of an illustrative string which utilizes the
2 relationship mechanism: .OWL:"string".

3 The "OWL" identifier identifies the ontology service and the "string"
4 specifies the specific string within the ontology service. Thus, the extended type
5 manager may access types supplied by ontology services.

6 Exemplary Process for Displaying Command Line Data

7 The present mechanism provides a data driven command line output. The
8 formatting and outputting of the data is provided by one or more cmdlets in the
9 pipeline of cmdlets. Typically, these cmdlets are included within the non-
10 management cmdlets described in conjunction with FIGURE 2 above. The
11 cmdlets may include a format cmdlet, a markup cmdlet, a convert cmdlet, a
12 transform cmdlet, and an out cmdlet.

13 FIGURE 19 graphically depicts exemplary sequences 1901-1907 of these
14 cmdlets within a pipeline. The first sequence 1901 illustrates the out cmdlet 1910
15 as the last cmdlet in the pipeline. In the same manner as described above for other
16 cmdlets, the out cmdlet 1910 accepts a stream of pipeline objects generated and
17 processed by other cmdlets within the pipeline. However, in contrast to most
18 cmdlets, the out cmdlet 1910 does not emit pipeline objects for other cmdlets.
19 Instead, the out cmdlet 1910 is responsible for rendering/displaying the results
20 generated by the pipeline. Each out cmdlet 1910 is associated with an output
21 destination, such as a device, a program, and the like. For example, for a console
22 device, the out cmdlet 1910 may be specified as out/console; for an internet
23 browser, the out cmdlet 1910 may be specified as out/browser; and for a window,
24 the out cmdlet 1910 may be specified as out/window. Each specific out cmdlet is
25

1 familiar with the capabilities of its associated destination. Locale information
2 (e.g., date & currency formats) are processed by the out cmdlet 1910, unless a
3 convert cmdlet preceded the out cmdlet in the pipeline. In this situation, the
4 convert cmdlet processed the local information.

5 Each host is responsible for supporting certain out cmdlets, such as
6 out/console. The host also supports any destination specific host cmdlet (e.g.,
7 out/chart that directs output to a chart provided by a spreadsheet application). In
8 addition, the host is responsible for providing default handling of results. The out
9 cmdlet in this sequence may decide to implement its behavior by calling other
10 output processing cmdlets (such as format/markup/convert/transform). Thus, the
11 out cmdlet may implicitly modify sequence 1901 to any of the other sequences or
12 may add its own additional format/output cmdlets.

13 The second sequence 1902 illustrates a format cmdlet 1920 before the out
14 cmdlet 1910. For this sequence, the format cmdlet 1920 accepts a stream of
15 pipeline objects generated and processed by other cmdlets within the pipeline. In
16 overview, the format cmdlet 1920 provides a way to select display properties and a
17 way to specify a page layout, such as shape, column widths, headers, footers, and
18 the like. The shape may include a table, a wide list, a columnar list, and the like.
19 In addition, the format cmdlet 1920 may include computations of totals or sums.
20 Exemplary processing performed by a format cmdlet 1920 is described below in
21 conjunction with FIGURE 20. Briefly, the format cmdlet emits format objects, in
22 addition to emitting pipeline objects. The format objects can be recognized
23 downstream by an out cmdlet (e.g., out cmdlet 1920 in sequence 1902) via the
24 extended type manager or other mechanism. The out cmdlet 1920 may choose to
25 either use the emitted format objects or may choose to ignore them. The out

1 cmdlet determines the page layout based on the page layout data specified in the
2 display information. In certain instances, modifications to the page layout may be
3 specified by the out cmdlet. In one exemplary process the out cmdlet may
4 determine an unspecified column width by finding a maximum length for each
5 property of a predetermined number of objects (e.g., 50) and setting the column
6 width to the maximum length. The format objects include formatting information,
7 header/footer information, and the like.

8 The third sequence 1903 illustrates a format cmdlet 1920 before the out
9 cmdlet 1910. However, in the third sequence 1903, a markup cmdlet 1930 is
10 pipelined between the format cmdlet 1920 and the out cmdlet 1910. The markup
11 cmdlet 1930 provides a mechanism for adding property annotation (e.g., font,
12 color) to selected parameters. Thus, the markup cmdlet 1930 appears before the
13 output cmdlet 1910. The property annotations may be implemented using a
14 "shadow property bag", or by adding property annotations in a custom namespace
15 in a property bag. The markup cmdlet 1930 may appear before the format cmdlet
16 1920 as long as the markup annotations may be maintained during processing of
17 the format cmdlet 1920.

18 The fourth sequence 1904 again illustrates a format cmdlet 1920 before the
19 out cmdlet 1910. However, in the fourth sequence 1904, a convert cmdlet 1940 is
20 pipelined between the format cmdlet 1920 and the out cmdlet 1910. The convert
21 cmdlet 1940 is also configured to process the format objects emitted by the format
22 cmdlet 1920. The convert cmdlet 1940 converts the pipelined objects into a
23 specific encoding based on the format objects. The convert cmdlet 1940 is
24 associated with the specific encoding. For example, the convert cmdlet 1940 that
25 converts the pipelined objects into Active Directory Objects (ADO) may be

1 declared as “convert/ADO” on the command line. Likewise, the convert cmdlet
2 1940 that converts the pipelined objects into comma separated values (csv) may be
3 declared as “convert/csv” on the command line. Some of the convert cmdlets
4 1940 (e.g., convert/XML and convert/html) may be blocking commands, meaning
5 that all the pipelined objects are received before executing the conversion.
6 Typically, the out cmdlet 1920 may determine whether to use the formatting
7 information provided by the format objects. However, when a convert cmdlet
8 1920 appears before the out cmdlet 1920, the actual data conversion has already
9 occurred before the out cmdlet receives the objects. Therefore, in this situation,
10 the out cmdlet can not ignore the conversion.

11 The fifth sequence 1905 illustrates a format cmdlet 1920, a markup cmdlet
12 1930, a convert cmdlet 1940, and an out cmdlet 1910 in that order. Thus, this
13 illustrates that the markup cmdlet 1930 may occur before the convert cmdlet 1940.

14 The sixth sequence 1906 illustrates a format cmdlet 1920, a specific convert
15 cmdlet (e.g., convert/xml cmdlet 1940’), a specific transform cmdlet (e.g.,
16 transform/xslt cmdlet 1950), and an out cmdlet 1910. The convert/xml cmdlet
17 1940’ converts the pipelined objects into an extended markup language (XML)
18 document. The transform/xslt cmdlet 1950 transforms the XML document into
19 another XML document using an Extensible Style Language (XSL) style sheet. The
20 transform process is commonly referred to as extensible style language
21 transformation (XSLT), in which an XSL processor reads the XML document and
22 follows the instructions within the XSL style sheet to create the new XML
23 document.

24 The seventh sequence 1907 illustrates a format cmdlet 1920, a markup
25 cmdlet 1930, a specific convert cmdlet (e.g., convert/xml cmdlet 1940’), a specific

1 transform cmdlet (e.g., transform/xslt cmdlet 1950), and an out cmdlet 1910.
2 Thus, the seventh sequence 1907 illustrates having the markup cmdlet 1930
3 upstream from the convert cmdlet and transform cmdlet.

4 FIGURE 20 illustrates exemplary processing 2000 performed by a format
5 cmdlet. The formatting process begins at block 2001, after the format cmdlet has
6 been parsed and invoked by the parser and pipeline processor in a manner
7 described above. Processing continues at block 2002.

8 At block 2002, a pipeline object is received as input to the format cmdlet.
9 Processing continues at block 2004.

10 At block 2004, a query is initiated to identify a type for the pipelined
11 object. This query is performed by the extended type manager as described above
12 in conjunction with FIGURE 18. Once the extended type manager has identified
13 the type for the object, processing continues at block 2006.

14 At block 2006, the identified type is looked up in display information. An
15 exemplary format for the display information is illustrated in FIGURE 21 and will
16 be described below. Processing continues at decision block 2008.

17 At decision block 2008, a determination is made whether the identified type
18 is specified within the display information. If there is no entry within the display
19 information for the identified type, processing is complete. Otherwise, processing
20 continues at block 2010.

21 At block 2010, formatting information associated with the identified type is
22 obtained from the display information. Processing continues at block 2012.

23 At block 2012, information is emitted on the pipeline. Once the
24 information is emitted, the processing is complete.
25

1 Exemplary information that may be emitted is now described in further
2 detail. The information may include formatting information, header/footer
3 information, and a group end/begin signal object. The formatting information may
4 include a shape, a label, numbering/bullets, column widths, character encoding
5 type, content font properties, page length, group-by-property name, and the like.
6 Each of these may have additional specifications associated with it. For example,
7 the shape may specify whether the shape is a table, a list, or the like. Labels may
8 specify whether to use column headers, list labels, or the like. Character encoding
9 may specify ASCII, UTF-8, Unicode, and the like. Content font properties may
10 specify the font that is applied to the property values that are display. A default
11 font property (e.g., Courier New, 10 point) may be used if content font properties
12 are not specified.

13 The header/footer information may include a header/footer scope, font
14 properties, title, subtitle, date, time, page numbering, separator, and the like. For
15 example, the scope may specify a document, a page, a group, or the like.
16 Additional properties may be specified for either the header or the footer. For
17 example, for group and document footers, the additional properties may include
18 properties or columns to calculate a sum/total, object counts, label strings for totals
19 and counts, and the like.

20 The group end/begin signal objects are emitted when the format cmdlet
21 detects that a group-by property has changed. When this occurs, the format cmdlet
22 treats the stream of pipeline objects as previously sorted and does not re-sort them.
23 The group end/begin signal objects may be interspersed with the pipeline objects.
24 Multiple group-by properties may be specified for nested sorting. The format
25 cmdlet may also emit a format end object that includes final sums and totals.

Turning briefly to FIGURE 21, an exemplary display information 2100 is in a structured format and contains information (e.g., formatting information, header/footer information, group-by properties or methods) associated with each object that has been defined. For example, the display information 2100 may be XML-based. Each of the afore-mentioned properties may then be specified within the display information. The information within the display information 2100 may be populated by the owner of the object type that is being entered. The operating environment provides certain APIs and cmdlets that allow the owner to update the display information by creating, deleting, and modifying entries.

FIGURE 22 is a table listing an exemplary syntax 2201-2213 for certain format cmdlets (e., format/table, format/list, and format/wide), markup cmdlets (e.g., add/markup), convert cmdlets (e.g., convert/text, convert/sv, convert/csv, convert/ADO, convert/XML, convert/html), transform cmdlets (e.g., transform/XSLT) and out cmdlets (e.g., out/console, out/file). FIGURE 23 illustrates results rendered by the out/console cmdlet using various pipeline sequences of the output processing cmdlets (e.g., format cmdlets, convert cmdlets, and markup cmdlets).

As described, the mechanism for obtaining and applying constraints in an interactive environment may be employed in an administrative tool environment. However, those skilled in the art will appreciate that the mechanism may be employed in various interactive environments.

Although details of specific implementations and embodiments are described above, such details are intended to satisfy statutory disclosure obligations rather than to limit the scope of the following claims. Thus, the invention as defined by the claims is not limited to the specific features described

1 above. Rather, the invention is claimed in any of its forms or modifications that
2 fall within the proper scope of the appended claims, appropriately interpreted in
3 accordance with the doctrine of equivalents.
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25